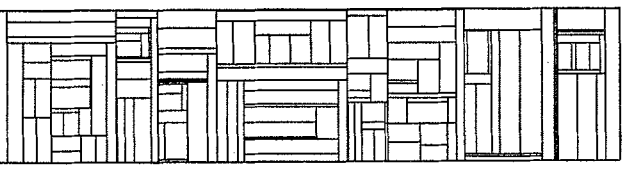
CONNECTION BETWEEN DIJKSTRA'S PREDICATE-TRANSFORMERS AND DENOTATIONAL CONTINUATION-SEMANTICS

by
Kurt Jensen

DAIMI PB-86
January 1978

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55



CONNECTION BETWEEN DIJKSTRA'S PREDICATE-TRANSFORMERS AND DENOTATIONAL CONTINUATION-SEMANTICS.

by

Kurt Jensen

Abstract

It is important to define and relate different semantic methods. In particular it is interesting to compare semantics used for program-verification to those aimed for program-execution. In this paper the intuitive background for a number of different semantics is given. They are all reformulated to the notation of denotational semantics and compared. It is shown that Dijkstra's weakest predicate theory is satisfied by a denotational continuation-semantics.

The present paper is an improved and shortened version of DAIMI PB-61.

1. Introduction

In this paper I shall investigate the connection between different semantic approaches. In [7] Hoare and Lauer argues for the need of such different semantics. A semantics useful for language-designers may not be useful for compiler-implementators nor for programmers using the language. Therefore it is of great importance to define and relate different semantic methods. It seems in particular interesting to compare semantics used for program-verification to those aimed for program-execution.

In this paper the intuitive background for a number of different semantics is revealed. It is a shortened version of [8] which contains detailed proofs of all mentioned theorems and uses a more complicated language containing declarations, non-deterministic commands and input/output-commands.

Section 2 defines an example-language, SNAIL. Section 3 summarises the work done by Hoare and Lauer in [7]. In section 4 this work is reformulated using the notation of denotational semantics. Section 5 introduces Dijkstra's weakest predicate-transformers. In section 6 our denotational semantics is augmented by command-continuations. It turns out that for Dijkstra's theory this contributes considerably to a more natural and understandable interpretation. It is shown that the predicate-transformer theory is satisfied (under a given interpretation) by the denotational continuation-semantics. Section 7 discusses the direction of further research on these topics. An appendix contains a formal definition of SNAIL using denotational continuation-semantics.

I want to express my gratitude to Brian Mayoh, Ole Lehrmann Madsen, Antoni Mazurkiewicz, Peter Mosses and Robin Milner. They all made useful contributions and comments for this work.

2. A simple language called SNAIL

This section gives an informal description of an example-language, SNAIL, used in the rest of this paper. A formal description using denotational continuation-semantics can be found in the appendix. Readers familiar with denotational semantics may choose to study the appendix and possibly skip the rest of this section.

By convention a word in capital letters denotes a domain (see Scott [17]) while the same word written in small letters (and possibly indexed or marked) denotes an element of the domain. As an example dop_1, $dop' \in DOP$ where DOP is the (syntactic) domain consisting of all Dyadic OPerators.

A SNAIL-program is simply the same as a SNAIL-command:

prog :: = com

A SNAIL-command is a block, a sequence of commands, an assignment, an IF-command, a WHILE-command or a DUMMY-command; all with the usual semantics. It should be noticed that SNAIL has no declarations.

com :: = BEGIN com END | com_1 ; com_2 | var:= exp |
IF exp THEN com_1 ELSE com_2 |
WHILE exp DO com | DUMMY

A SNAIL-expression is either: an expression enclosed by parentheses, a constant, a variable, a monadic (prefixed) operator used on an expression or a dyadic (infix) operator used on two expressions.

exp :: = (exp) | con | var |
mop exp | exp_1 dop exp_2

The syntactic domains of programs, commands, expressions and variables are denoted PROG, COM, EXP and VAR respectively.

3. Four different semantics - Hoare and Lauer

In [7] Hoare and Lauer formulate four different semantics for a language similar to SNAIL. These semantics range from an extreme constructive approach depending on an (abstract) machine to an extreme implicit approach where semantics are defined via a logical deduction system using axioms and rules of inference. In [9] Lauer extends these semantics to a richer language containing nondeterminism, procedures and declarations. In this section I shall give a very brief explanation of the intuitive contents of these semantics without giving the actual definitions.

Interpretive Model

This approach describes how an abstract machine executes a program step by step. It is a simplified version of the VDL definition-language developed by IBM Laboratory Vienna (see MARCOTTY, LEDGARD and BOCHMANN [12]).

A machine configuration is a pair $(s, com) \in S \times COM$ where s is a member of the domain of memory states and com is the control state describing the commands remaining to be executed. The state transformations performed by the abstract machine is defined by a transition function

$$next \in [S \times COM \to S \times COM].$$

The result of a SNAIL-computation is the memory state obtained by iterated use of the function next until the control state becomes empty:

$$result \in [S \times PROG \to S]$$

$$result(s, prog) = \begin{cases} s & \text{iff } prog = empty \\ result(next(s, prog)) & \text{else} \end{cases}$$

Computational Model

This approach differs from the previous one in that it discards the notion of control state. Now an initial state and a program is mapped in a sequence of memory states representing the entire computation:

$$comp \in [S \times PROG \to S^*]$$

The definition is recursive.

Relational Theory

We now move into the implicit approaches by discarding all intermediate steps in the computation. This can be compared with the difference between a procedure specifying in detail every step in an evaluation-algorithm and a (mathematical) function merely giving a relation between values in definition-domain and range without prescribing an actual evaluation-method. For a further discussion of this see Scott [17] and Strachey [19].

In this approach each command is equipped with a relation on S × S where the intuitive meaning of $s_1 R_{com} s_2$ ((s_1, s_2) element of the relation associated with com) is that execution of com brings the abstract machine from initial state s_1 to final state s_2. The relations are defined by axioms ; theorems derived from these axioms are intended to be valid assertions about memory states before and after execution of a program.

Deductive Theory

We now introduce a further abstraction by discarding memory states and only handling predicates describing properties of memory states. This approach has been used by Hoare [6]. As in the relational theory commands will be associated with relations, but now these relations will be on PRED × PRED where PRED is the domain of first order logical predicates defined on domain S, and with elements of VAR as free variables. The relations are defined by axioms and deduction rules and

$$\{ pred_1 \} \ com \ \{ pred_2 \}$$

has the intuitive meaning that execution of com with an initial state satisfying $pred_1$ will either result in a final state satisfying $pred_2$ or fail to terminate. We say that com is <u>partially correct</u> with respect to $pred_1$ and $pred_2$.

Connection between the 4 semantics

Theorem 1

The interpretive and computational models are equivalent:

$$I\,last \circ comp = result$$

□

where last is a function yielding the last item of a non-empty finite sequence and undefined for empty or infinite arguments (and $f \circ g(x)$ means $f(g(x))$.

Theorem 2

The relational theory is satisfied by the computational model under the interpretation:

$$s_1 \; R_{prog} \; s_2 \quad \equiv \quad s_2 = last\,(\,comp\,(s_1\,,\,prog))$$

□

Theorem 3

The deductive theory is conservative with respect to the relational theory under the interpretation:

$$\{\,pred_1\,\}\;com\;\{\,pred_2\,\} \equiv$$
$$\forall\, s_1\; s_2\;\; [pred_1\;(s_1) \wedge s_1\; R_{com}\; s_2 \Rightarrow pred_2(s_2)]$$

□

By conservatism is meant that any theorem deducible in the deductive theory also has a deduction in the relational theory.

Corollary

The interpretive and computational models are equivalent and the relational and deductive theories are satisfied under these two models (using the given interpretations).

□

4. Four different semantics - a reformulation

In this section I shall reformulate Hoare and Lauer's ideas using the notation of denotational semantics (see Scott, Strachey and Wadsworth [17, 18, 19, 20] or Tennent [21]). It turns out that such a change of notation in many aspects make the theorems and proofs more straightforward and understandable.

Interpretive Model

In the original formulation meaning is assigned to programs by a function

$$\text{result} \in [S \times \text{PROG} \to S]$$

In denotational semantics (without continuations) it is common practice to use a function

$$\mathcal{C}_i \in [\text{COM} \to [S \to S]]$$

Since the above two domains are isomorphic (remember PROG = COM) it is tempting to substitute $\mathcal{C}_i \llbracket \text{com} \rrbracket s$ everywhere for result (s, com).

This gives the following semantics

$$\mathcal{C}_i \llbracket \underline{\text{BEGIN}}\ \text{com}\ \underline{\text{END}} \rrbracket s = \mathcal{C}_i \llbracket \text{com} \rrbracket s$$

$$\mathcal{C}_i \llbracket \text{com}_1 ; \text{com}_2 \rrbracket s = \mathcal{C}_i \llbracket \text{com}_2 \rrbracket (\mathcal{C}_i \llbracket \text{com}_1 \rrbracket s)$$

$$\mathcal{C}_i \llbracket \text{var:=exp} \rrbracket s = \text{ASSIGN}(\text{var}, \mathcal{E}_i \llbracket \text{exp} \rrbracket s)s$$

$$\mathcal{C}_i \llbracket \underline{\text{IF}}\ \text{exp}\ \underline{\text{THEN}}\ \text{com}_1\ \underline{\text{ELSE}}\ \text{com}_2 \rrbracket s = \mathcal{E}_i \llbracket \text{exp} \rrbracket s \to \mathcal{C}_i \llbracket \text{com}_1 \rrbracket s, \mathcal{C}_i \llbracket \text{com}_2 \rrbracket s$$

$$\mathcal{C}_i \llbracket \underline{\text{WHILE}}\ \text{exp}\ \underline{\text{DO}}\ \text{com} \rrbracket s = \mathcal{E}_i \llbracket \text{exp} \rrbracket s \to \mathcal{C}_i \llbracket \underline{\text{WHILE}}\ \text{exp}\ \underline{\text{DO}}\ \text{com} \rrbracket (\mathcal{C}_i \llbracket \text{com} \rrbracket s), s = (Y(\lambda F.\ \lambda s'.\ \mathcal{E}_i \llbracket \text{exp} \rrbracket s' \to F(\mathcal{C}_i \llbracket \text{com} \rrbracket s'), s'))s$$

$$\mathcal{C}_i \llbracket \text{DUMMY} \rrbracket s = s$$

$\mathcal{E}_i$ is a semantic function giving meaning to expressions ($\mathcal{E}_i \in [\text{EXP} \to [S \to V]]$). V is the domain of expression values. Expressions have no side-effects and their evaluations always terminate. ASSIGN is an auxiliary function with functionality

$$\text{ASSIGN} \in [\text{VAR} \times V \to [S \to S]]$$

ASSIGN is defined analogous with UPDATE (see appendix).

Y is the least fixpoint operator on domain S.

" → " is the conditional operator defined by

$$v \rightarrow s_1 \ , \ s_2 \ = \begin{cases} s_1 & \text{iff } v = TT \\ s_2 & \text{iff } v = FF \\ \perp_S & \text{else} \end{cases}$$

(TT and FF are supposed to be elements of v representing TRUE and FALSE respectively).

It is easily seen that this is a "natural" denotational semantics for SNAIL.

Computational Model

This model is treated analogously. We substitute $C_a \ [\![com]\!]$ s for comp(s, com) which gives C_a functionality

$$C_a \in [COM \rightarrow [S \rightarrow S^*]].$$

However, it turns out that this gives a rather lengthy and inelegant equation for $C_a \ [\![com_1 \ ; \ com_2]\!]$. To avoid this we "lift" C_a to a function

$$C_c \in [COM \rightarrow [S^* \rightarrow S^*]]$$

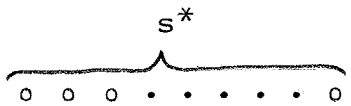
defined by

$$C_c \ [\![com]\!] \ s^* = s^* \ || \ tail \ (C_a \ [\![com]\!] \ last \ (s^*))$$

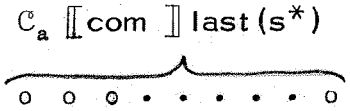
where $s^* \in S^*$, " || " is the append operator and tail is a function yielding its argument (a sequence) with initial element removed and undefined for empty argument. (last is defined in connection with theorem 1).

This apparently complex definition has the following straightforward intuitive visualization:

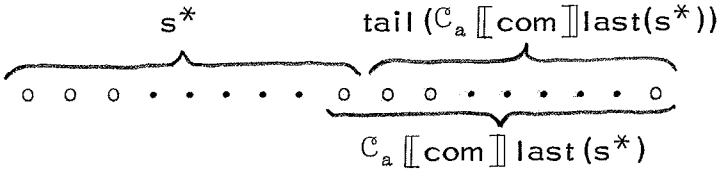
Initial state sequence



$C_a \ [\![com]\!]$ applied to last (s^*)



Concatenation – omitting one of the two occurences of last (s^*)



We now get $\mathcal{C}_c [\![com_1 ; com_2]\!] s^* = \mathcal{C}_c [\![com_2]\!] (\mathcal{C}_c [\![com_1]\!] s^*)$

Again it could be seen by a more detailed inspection that the obtained function $\mathcal{C}_c$ is a "natural" denotational semantics for SNAIL

Relational and Deductive Theory

The two theories are defined by axioms and deduction rules. These are not affected by the reformulation on the previous pages; only the interpretations of the theories are altered.

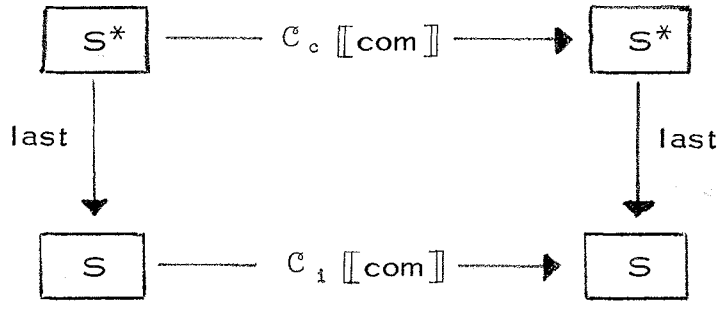
Connection between the 4 semantics

Theorem 4

The reformulated and computational models are equivalent:

$$last (\mathcal{C}_c [\![com]\!] s^*) = \mathcal{C}_i [\![com]\!] last(s^*)$$ □

This can also be expressed as commutation of the following diagram (for all com ∈ COM):



Theorem 5

The relational theory is satisfied by the reformulated interpretive model under the interpretation:

$$s_1 R_{com} s_2 \quad \equiv \quad s_2 = \mathcal{C}_i [\![com]\!] s_1$$

□

Theorem 6

The deductive theory is satisfied by the reformulated interpretive model under the interpretation:

$$\{ pred_1 \} \; com \; \{ pred_2 \}$$
$$\equiv \forall s \; [pred_1 \; (s) \wedge \mathcal{C}_i \; [\![com]\!] s \neq \perp_s$$
$$\Rightarrow pred_2 \; (\mathcal{C}_i \; [\![com]\!] s) \;]$$

□

It should be noticed that in this reformulation the two theories are interpreted directly in terms of the interpretive model and not via the computational model.

5. Predicate-transformer theory

In [1, 2, 3, 4] Dijkstra presents a semantic theory based on predicate-transformers. Its ideas are very closely related to those represented by Hoares deductive theory, but there are 2 main differences:

1) Dijkstra demands <u>total</u> correctness (i.e. termination) while Hoare only demands <u>partial</u> correctness

2) Dijkstra demands a <u>sufficient</u> and <u>necessary</u> (weakest) precondition while Hoare only demands <u>sufficiency</u>

Dijkstra defines (by axioms) a function

$$wp \in [COM \times PRED \rightarrow PRED \;].$$

The intuitive meaning of wp(com, $pred_1$) is the weakest precondition (predicate), $pred_2$, which guarantees that execution of com with an initial state satisfying $pred_2$ terminates (without error) with a final state satisfying $pred_1$.

The semantics of SNAIL in this theory is expressed by the following axioms

WP 1 : $wp(\underline{BEGIN}\ com\ \underline{END}, pred) \equiv wp(com, pred)$

WP 2 : $wp(com_1 ; com_2, pred) \equiv wp(com_1, wp(com_2, pred))$

WP 3 : $wp(var := exp, pred) \equiv pred \mid \begin{matrix} var \\ exp \end{matrix}$

WP 4 : $wp(\underline{IF}\ exp\ \underline{THEN}\ com_1\ \underline{ELSE}\ com_2, pred)$

$\equiv (exp \wedge wp(com_1, pred)) \vee (\neg exp \wedge wp(com_2, pred))$

WP 5 : $wp(\underline{WHILE}\ exp\ \underline{DO}\ com, pred) \equiv (\exists i \geq 0 : H_i(pred))$

where $H_0(pred) \equiv (\neg exp \wedge pred)$

$H_i(pred) \equiv wp(\underline{IF}\ exp\ \underline{THEN}\ com\ \underline{ELSE}\ \underline{DUMMY}, H_{i-1}(pred))\quad (i \geq 1)$

WP 6 : $wp(\underline{DUMMY}, pred) \equiv pred$

where $pred \mid \begin{matrix} var \\ exp \end{matrix}$ denotes pred with exp substituted for all free occurences of var (expressions are evaluated without sideeffects).

Theorem 7

The predicate-transformer theory is satisfied by the reformulated interpretive model under the interpretation:

$$wp(com, pred)\, s \equiv pred(\mathbb{C}_i [\![com]\!] s)$$

with the convention that all predicates map $\perp_S$ to FALSE.

□

6. Predicate-transformer theory and continuation-semantics

It is now the time to introduce command-continuations and investigate how this affects the interpretation of the 3 theories. For the relational and deductive theories it turns out that this adds very little to the flexibility and understanding of the theories, but when the predicate-transformer theory is considered the situation is quite different. Here the use of command-continuations is very convenient and a much more natural and straightforward interpretation is achieved.

We make the assumption that all our predicates are <u>continuous</u> functions from the domain of memory states, S, into the domain of truthvalues TF. (TF = { false, true } with ordering false $\subseteq$ true). Then PRED = [S $\to$ TF].

We define the domain of command-continuations by CC = [S $\to$ A] where A is an answer domain assumed to contain TF. Then PRED $\subseteq$ CC.

Moreover we define a denotational continuation-semantics

$$\mathcal{C} \in [\,COM \to [CC \to CC]\,]$$

by the equation

$$\mathcal{C}\,[\![com]\!]\,cc\,s = cc\,(\mathcal{C}_1\,[\![com]\!]\,s)$$

It can be seen that this definition of $\mathcal{C}$ is equivalent to that in the appendix.

The original interpretation:

$$wp(com,\ pred)s = pred\ (\mathcal{C}_1\,[\![com]\!]\,s)$$

then becomes

$$wp(com,\ pred)\,s = \mathcal{C}\,[\![com]\!]\,pred\,s$$

or

$$wp(com,\ pred) = \mathcal{C}\,[\![com]\!]\ pred$$

where we moreover assume that all predicates are strict (i.e. they map $\bot_S$ on $\bot_{TF}$ = FALSE).

Thus we have established a close connection between the two functions:

$$wp \in [\,COM \times PRED \to PRED\,]$$
$$\mathcal{C} \in [\,COM \to [CC \to CC]\,]$$

where it should be remembered that PRED $\subseteq$ CC.

It gives a much more elegant interpretation of Dijkstra's theory for predicate-transformers. Moreover it shows a connection between program-execution represented by $\mathcal{C}$ and program-verification represented by wp. This idea, originally due to Brian Mayoh, has very interesting aspects as it suggests a possibility for combining verification- and execution-systems thus eliminating the problem of proving consistency of a verification-system with respect to an actual implementation.

To use predicates as command-continuations is technically nothing but choosing the answer domain, A, to be (or at least contain) the domain of truthvalues TF. This is only one possibility among many. Compared to the use of memory states as answer domain it reveals one outstanding difference. Using memory states we define a function mapping initial-states into final-states while using predicates we define a transformation from post-conditions to weakest pre-conditions . This is very much the same as the difference between relational theory and deductive theory. The former deals explicitly with memory states; the latter only implicitly via predicates.

Theorem 8

The predicate-transformer theory is satisfied by the denotational continuation-semantics represented by $\mathbb{C}$ (see appendix) under the interpretation:

$$wp(com,\ pred) \equiv \mathbb{C}[\![com]\!]pred$$ □

Proof for theorem 8

The proof is done by structural induction on commands:

WP1 :

$$wp(\underline{BEGIN}\ com\ \underline{END},\ pred)$$
$$\equiv \mathbb{C}[\![\underline{BEGIN}\ com\ \underline{END}]\!]pred$$
$$\equiv \mathbb{C}[\![com]\!]pred$$
$$\equiv wp(com,\ pred)$$

WP2 :

$$wp(com_1\ ;\ com_2\ ,\ pred)$$
$$\equiv \mathbb{C}[\![com_1\ ;\ com_2]\!]pred$$
$$\equiv \mathbb{C}[\![com_1]\!]\{\ \mathbb{C}[\![com_2]\!]pred\}$$
$$\equiv wp(com_1\ ,\ \mathbb{C}[\![com_2]\!]pred)$$
$$\equiv wp(com_1\ ,\ wp(com_2\ ,\ pred))$$

WP3 : $wp(var := exp, pred)$

$\equiv \mathcal{C}[\![var := exp]\!] pred$

$\equiv \mathcal{E}[\![exp]\!] \{\lambda v.\ UPDATE(var, v)\ pred\}$

$\equiv (pred \mid^{var}_{exp})$

by the definition of UPDATE (see appendix) and under the assumption that expressions are evaluated without side-effects.

WP4 : $wp(\underline{IF}\ exp\ \underline{THEN}\ com_1\ \underline{ELSE}\ com_2, pred)$

$\equiv \mathcal{C}[\![\underline{IF}\ exp\ \underline{THEN}\ com_1\ \underline{ELSE}\ com_2]\!] pred$

$\equiv \mathcal{E}[\![exp]\!] \{COND(\mathcal{C}[\![com_1]\!] pred, \mathcal{C}[\![com_2]\!] pred)\}$

$\equiv ((exp \wedge \mathcal{C}[\![com_1]\!] pred) \vee (\neg exp \wedge \mathcal{C}[\![com_2]\!] pred))$

$\equiv ((exp \wedge wp(com_1, pred)) \vee (\neg exp \wedge wp(com_2, pred)))$

where "exp" ("$\neg$ exp") in the last two lines denotes the predicate yielding TRUE for exactly those states where exp $\in$ EXP evaluates to TRUE (FALSE).

WP5 : $wp(\underline{WHILE}\ exp\ \underline{DO}\ com, pred)$

$\equiv \mathcal{C}[\![\underline{WHILE}\ exp\ \underline{DO}\ com]\!] pred$

$\equiv Y(\lambda cc'.\ \mathcal{E}[\![exp]\!] \{COND(\mathcal{C}[\![com]\!] cc', pred)\})$

$\equiv Y(\lambda cc'.\ ((exp \wedge \mathcal{C}[\![com]\!] cc') \vee (\neg exp \wedge pred)))$

$\equiv Y(\lambda pred'.(((exp \wedge wp(com, pred')) \vee (\neg exp \wedge pred))))$

$\equiv Y(F)$

where $F = \lambda pred'.(((exp \wedge wp(com, pred')) \vee (\neg exp \wedge pred)))$

We first show by induction that for all $i \geq 0$:

(*) $\neg exp \wedge H_i(pred) \equiv \neg exp \wedge pred$

where H_i is defined in wp 5 (section 5).

case i = 0 : $\neg exp \wedge H_0(pred)$

$\equiv \neg exp \wedge (\neg exp \wedge pred)$

using wp 5

$\equiv \neg exp \wedge pred$

case i $\geq$ 1 : $\neg exp \wedge H_i(pred)$

$\equiv \neg exp \wedge wp(\underline{IF}\ exp\ \underline{THEN}\ com\ \underline{ELSE}\ \underline{DUMMY}, H_{i-1}(pred))$

$\equiv \neg exp \wedge ((exp \wedge wp(com, H_{i-1}(pred))) \vee$

$(\neg exp \wedge wp(\underline{DUMMY}, H_{i-1}(pred))))$

$\equiv \neg exp \wedge H_{i-1}(pred)$

using wp 5, wp 3 and wp 6

$\equiv \neg exp \wedge pred$

by the inductional hypothesis.

We then have

$H_i(pred) \equiv wp(\underline{IF}\ exp\ \underline{THEN}\ com\ \underline{ELSE}\ \underline{DUMMY}, H_{i-1}(pred))$

$\equiv (exp \wedge wp(com, H_{i-1}(pred))) \vee (\neg exp \wedge H_i(pred))$

using wp 5, wp 3 and wp 6

$\equiv (exp \wedge wp(com, H_{i-1}(pred))) \vee (\neg exp \wedge pred)$

using (*)

$\equiv F(H_{i-1}(pred))$

and

$H_0(pred) \equiv \neg exp \wedge pred$

$\equiv F(\perp_{PRED})$

We thus conclude that the sequence

$\{ H_i(\text{pred}) \}_{i \geq 0}$ is exactly the sequence

$\{ F^i(\bot_{PRED}) \}_{i \geq 1}$ used when calculating

the minimal fixpoint $Y(F)$ and hence

$$Y(F) \equiv \exists i \geq 0 : H_i(\text{pred})$$

which finishes the proof.

(It should be mentioned that this proof would have been much simpler if wp(WHILE exp DO com, pred) had been defined using the minimal fixpoint operator, but I wanted to use Dijkstra's original definition for the WHILE-command).

WP6 : wp (DUMMY, pred)

$\equiv$ $\mathbb{C}$ [[DUMMY]] pred

$\equiv$ pred

□

REFERENCES

1. DIJKSTRA, E.W.: A simple axiomatic basis for programming language constructs, Indagationes Mathematicae, 36 (1974) 1-15 (EWD 372).

2. DIJKSTRA, E.W.: Guarded commands, non determinacy and calculus for the derivation of programs, Comm. ACM, 18, 453-457 (aug. 1975)

3. DIJKSTRA, E.W.: Sequencing primitives revisited, Technical University Eindhoven, The Netherlands (1973) (EWD 398 + EWD 399).

4. DIJKSTRA, E.W.: A discipline of programming, Prentice Hall Inc. Englewood Cliffs, New Jersey, (1976).

5. DONAHUE, J.E.: The mathematical semantics of axiomatically defined programming language constructs, Colloques IRIA, Arc et Senans 1-3 juillet 1975, 353-367.

6. HOARE, C.A.R.: An axiomatic basis for computer programming, Comm. ACM, 12 576-580 (1967)

7. HOARE, C.A.R. and LAUER, P.: Consistent and complementary formal theories of the semantics of programming languages, Acta Inform. 3, 135-153 (1974) by Springer-Verlag

8. JENSEN, K.: An investigation into different semantic approaches, DAIMI PB-61, Dept. of Computer Science, Ny Munkegade, 8000 Aarhus C, Denmark

9. LAUER, P.: Consistent formal theories of the semantics of programming languages, IBM Laboratory Vienna, TR 25.121 (Nov. 1971)

10. LEVIN, M.: Mathematical logic for computer scientists, Massachusetts Institute of Technology, MAC TR-131 (June 1974)

11. LIGLER, G.: Surface properties of programming language constructs, Colloques IRIA, Arc et Senans 1-3 juillet 1975, 299-323

12. MARCOTTY, M., LEDGARD, H.F. and BOCHMANN, G.V.: A Sampler of Formal Definitions, Comp. Surveys, Vol. 8, No. 2, 191-276 (June 1976)

13. MOSSES, P.: The mathematical semantics of Algol 60, Technical Monograph PRG-12, Oxford University Computing Laboratory, (Jan. 1974)

14. MOSSES, P.: Mathematical Semantics and Compiler Generation, D. Phil. Thesis, University of Oxford, (1975)

15. MOSSES, P.: The Semantics of Semantic Equations. Proc. Symp. on Math. Found. Comp. Sci., Jadwisin 1974, Lect. Notes Comput. Sci. 28, 409-422, Springer Verlag, (1975)

16. MOSSES, P.: Compiler Generation using Denotational Semantics, Proc. Symp. on Math. Found. Com. Sci., Gdańsk 1976, Lect. Notes Comput. Sci. 45, 436-441, Springer Verlag, (1976)

17. SCOTT, D.: Outline of mathematical theory of computation, Proc. of the fourth anual Princeton conference on information science and systems, 169-176, and Technical Monograph PRG-2, Oxford University Computing Laboratory, (Nov. 1970)

18. SCOTT, D. and STRACHEY, C.: Toward a mathematical semantics for computer languages, Proc. of the symposium on computers and automata, Polytechnic Institute of Brooklyn, and Technical Monograph PRG-6, Oxford University Computing Laboratory, (Aug. 1971)

19. STRACHEY, C.: The varieties of programming language, Proc. of the international computing symposium 222-233, Cini Foundations Venice, and Technical Monograph PRG-10, Oxford University Computing Laboratory, (March 1973)

20. STRACHEY, C. and WADSWORTH C.P.: Continuations. A mathematical semantics for handling full jumps, Technical Monograph PRG-11, Oxford University Computing Laboratory, (Jan. 1974)

21. TENNENT, R.D.: The Denotational Semantics of Programming Languages, Comm. ACM, 19, 437-453, (1976)

APPENDIX

Formal Description of SNAIL

This appendix contains a formal description of the syntax and semantics for SNAIL. The syntax is described in a BNF-like notation, while semantics are described using denotational semantics (see Scott, Strachey and Wadsworth [17, 18, 19, 20] or Tennent [21]). The structure of this description is taken from Ligler [11] but some changes have been made to improve readability by using mnenomics instead of the usual Greek letters. By convention a word in capital letters denotes a domain (see Scott [17]) while the same word written in small letters (and possibly indexed or marked) denotes an element of the domain. As an example dop_1, dop' $\in$ DOP where DOP is the (syntactic) domain consisting of all Dyadic OPerators.

Syntactic domains

PROGrams	
COMmands	CONstants
EXPressions	Monadic OPerators
VARiables	Dyadic OPerators

(These are all simple, flat domains)

Semantic domains

Values	
Answers	
States	[VAR → V]
Command Continuations	[S → A]
Expression Continuations	[V → CC]

V is a simple, flat and implementation dependent domain assumed to contain TRUE and FALSE. A is a compound domain denoting the possible answers (results) of a computation in SNAIL.

Syntax

prog :: = com

com :: = BEGIN com END | com_1 ; com_2 | var : = exp |

IF exp THEN com_1 ELSE com_2 | WHILE exp DO com | DUMMY

exp :: = (exp) | con | var

mop exp | exp_1 dop exp_2

VAR, CON, MOP and DOP are implementation defined sets of symbols.

Semantic functions

$\mathcal{P}$: [PROG → A]
$\mathcal{C}$: [COM → [CC → CC]]
$\mathcal{E}$: [EXP → [EC → CC]]
$\mathcal{K}$: [CON → V]
$\mathcal{M}$: [MOP → [V → V]]
$\mathcal{D}$: [DOP → [V × V → V]]

where $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{E}$ give meaning to programs, commands and expressions respectively. $\mathcal{K}$, $\mathcal{M}$ and $\mathcal{D}$ are implementation dependent and give meaning to the elements of CON, MOP and DOP.

Semantic equations

$$\mathcal{P}[\![com]\!] = \mathcal{C}[\![com]\!]\, cc_{in}\, s_{in}$$

where $cc_{in} = I_S = \lambda s.\ s$

$s_{in} = \perp_S = \lambda\, var.\ \perp_V$

$$\mathcal{C}[\![\underline{\text{BEGIN}}\ \text{com}\ \underline{\text{END}}]\!]\,cc = \mathcal{C}[\![\text{com}]\!]\,cc$$

$$\mathcal{C}[\![\text{com}_1\ ;\ \text{com}_2]\!]\,cc = \mathcal{C}[\![\text{com}_1]\!]\,\{\mathcal{C}[\![\text{com}_2]\!]\,cc\}$$

$$\mathcal{C}[\![\text{var}:=\text{exp}]\!]\,cc = \mathcal{E}[\![\text{exp}]\!]\,\{\lambda v.\ \text{UPDATE}(\text{var}, v)\,cc\}$$

$$\mathcal{C}[\![\underline{\text{IF}}\ \text{exp}\ \underline{\text{THEN}}\ \text{com}_1\ \underline{\text{ELSE}}\ \text{com}_2]\!]\,cc$$
$$= \mathcal{E}[\![\text{exp}]\!]\,\{\text{COND}\,(\mathcal{C}[\![\text{com}_1]\!]\,cc, \mathcal{C}[\![\text{com}_2]\!]\,cc)\}$$

$$\mathcal{C}[\![\underline{\text{WHILE}}\ \text{exp}\ \underline{\text{DO}}\ \text{com}]\!]\,cc$$
$$= Y(\lambda\, cc'.\ \mathcal{E}[\![\text{exp}]\!]\,\{\text{COND}\,(\mathcal{C}[\![\text{com}]\!]\,cc', cc)\})$$

$$\mathcal{C}[\![\underline{\text{DUMMY}}]\!]\,cc = cc$$

$$\mathcal{E}[\![\ (\text{exp})\ ]\!]\,ec = \mathcal{E}[\![\text{exp}]\!]\,ec$$

$$\mathcal{E}[\![\text{con}]\!]\,ec = ec(\mathcal{K}[\![\text{con}]\!])$$

$$\mathcal{E}[\![\text{var}]\!]\,ec = \lambda s.(ec\,(\text{CONTENTS}\,(\text{var})\,s)\,s)$$

$$\mathcal{E}[\![\text{mop exp}]\!]\,ec = \mathcal{E}[\![\text{exp}]\!]\,\{\lambda v.\ ec(\mathbb{m}(\text{mop})\,v)\}$$

$$\mathcal{E}[\![\text{exp}_1\ \text{dop}\ \text{exp}_2]\!]\,ec$$
$$= \mathcal{E}[\![\text{exp}_1]\!]\,\{\lambda v_1.\ \mathcal{E}[\![\text{exp}_2]\!]\,\{\lambda v_2.\ ec(\mathbb{D}(\text{dop})\,(v_1, v_2))\}\}$$

Auxilliary functions

$$\text{COND} \in [CC \times CC \to [V \to CC]]$$

$$\text{COND}(cc_1, cc_2)(v) = v \to cc_1, cc_2 = \begin{cases} cc_1 & \text{iff } v = \text{TRUE} \\ cc_2 & \text{iff } v = \text{FALSE} \\ \perp_{CC} & \text{else} \end{cases}$$

$$Y \in [[CC \to CC] \to CC]$$

$$Y(f) = \lim_{n\to\infty} f^n(\perp_{CC})$$

$$UPDATE \in [VAR \times V \to [CC \to CC]]$$

$$UPDATE(var, v)\ cc\ s = cc(s') \text{ where}$$

$$s'(var') = \begin{cases} s(var') & \text{iff } var' \neq var \\ v & var' = var \end{cases}$$

$$CONTENTS \in [VAR \to [S \to V]]$$

$$CONTENTS\ (var)\ (s) = s(var)$$